

Overview

This Java program is designed to do computations with modular lattices. In particular, it is intended to find a normal form for a modular lattice with a given presentation by generators and relations. This is known to be impossible in many cases, but even in these cases this program may be able to produce useful information.

The main data structures of the program are the element list and the join/meet table. Elements are numbered in the order they are created, starting with the number 0. The user must create the first few elements to get the program started; for example, to compute the free modular lattice on three generators, the user creates the three generators. The join meet/table has as many rows and columns as there are elements in the element list. In position (i, j) is stored the join of element i and element j , if $i \leq j$, or their meet if $i \geq j$. The user may need to fill in some of the squares by hand to indicate that the lattice satisfies relations. The program fills the table, starting at position $(0, 0)$, followed by $(1, 0)$, $(0, 1)$, $(1, 1)$, $(2, 0)$, $(2, 1)$, $(0, 2)$, etc., alternately adding rows and columns to the filled area. For each square it visits, it attempts to prove that the relevant join or meet is equal to one of the existing elements. If it fails to do so, it creates a new element. (The computation will sometimes fail even when the join or meet really is equal to an existing element. So the objects that I have been calling “elements” would be more accurately called “terms”, because two of them may represent the same element of the lattice being computed.) If the table is filled, that means that all the elements have been found.

One feature of this program is that when filling a square, it does not stop computing when it finds the answer. In the course of these continued computations, it sometimes discovers that two of the elements are in fact equal. The program keeps track of this information using a data structure that represents an equivalence relation on the set of elements. Whenever the user commands, the program will discard all but one element of each equivalence class, and renumber the remaining elements.

So far I have used the words “lattice”, “element”, and “term” ambiguously. Henceforth, when they are capitalized they will denote Java classes, and when uncapitalized they will denote mathematical objects.

Notes

1. This program is not so good from a big- O perspective, and 10 years ago it would probably have run too slowly to be useful, but now, running on a Sun workstation, it can compute the free modular lattice on three generators in about 2 seconds, and can process hundreds of elements in a couple minutes.

2. The classes described here do not constitute a stand-alone program, because none of them has a `main()` method. The user must write code to create a `ModLattice`, fill in the information needed to begin the computation, and pass the `ModLattice` to the `start()` method in class `Driver`. I can provide

examples showing how this is done. In particular, I have written a class `Free` with a `main()` method that starts the computation of a free modular lattice on any number of generators. One could get by with just this, because the program has the ability to impose relations, and any modular lattice is a quotient of a free modular lattice.

Data Structures

Class `ModLattice`

This class is the core of the program. `ModLattice` has the following fields:

`protected Element [] elements`: The list of `Elements`; allows them to be accessed by number.

`protected Element [][] table`: The join/meet table. Note that it stores `Elements`, although it could also have been made to hold their numbers. Since each `Element` stores its number, it works either way.

`protected int currentSize`: The number of rows and columns in table. It is not really determined by the number of `Elements`, because it would be inefficient to resize table every time a new `Element` is created. `currentSize` is also the length of `elements`.

`protected int elementCount`: The number of `Elements` that have been created.

`protected EqRel myEqRel`: The equivalence relation.

`protected int currentRow, currentColumn`: These store the position of the next square of `table` to be filled.

`protected int auto`: The number of automorphisms used. This program has the capability to use automorphisms in its computations. Any finite subset of the lattice's automorphism group that is closed under inverses may be used, but the identity should not be used because it does not help. The automorphisms are numbered from 0 to $(\text{auto} - 1)$.

`protected int [] inverse`: An array of length `auto`. In the i th position it holds the number of the inverse of the i th automorphism. This array must be created by the user before constructing the `ModLattice`. If the lattice has no nontrivial automorphisms, or the user does not wish to use them, the user supplies an array of length 0.

Class `SelfDualModLattice`

This class is a subclass of `ModLattice`. It provides the ability to use duality in the computations. It has no extra fields.

Class `Term`

This class has the following fields:

`final char op`: Indicates what type of term this is. If `op` is 'v' (lowercase V), then the term is a join; if `op` is '^' (shift-6 on most keyboards), then the term is a meet. If `op` is any other character, then the term is a generator.

(Do not use whitespace characters for generators—this will interfere with the method that reads a lattice from a file.) In the code, generators are usually called variables. The characters '0' and '1' are special; a `Term` with `op` '0' (resp. '1') is assumed to represent the least (resp. greatest) element.

`final Element arg1, arg2`: These are the `Elements` that this `Term` is a join or meet of. Note only binary `Terms` are allowed. These fields are used only if `op` is 'v' or '^'.

The parameters required to construct a `Term` are the `op`, plus `arg1` and `arg2` when relevant.

Class `Element`

This class has the following fields:

`final ModLattice myLattice`: The `ModLattice` that this `Element` is in. If one wants the same `Element` in more than one `ModLattice`, it must be copied.
`Term myTerm`: The `Term` that this `Element` was defined with.
`Term [] altTerms`: An array that holds other `Terms` that have been found equal to this `Element`.
`int altsUsed`: The number of entries in `altTerms`. Its maximum value is the constant `ALTS`.
`final int myNumber`: Indicates the position of this `Element` in the `ModLattice`'s list of `Elements`.
`final int conjCount`: Number of conjugates; equal to `myLattice.auto`.
`Element [] conjugates`: Array of length `conjCount` that holds this `Element`'s conjugates.
`Element dual`: Dual of this `Element`; used only if this `Element` is in a `SelfDualModLattice`.

The parameters required to construct an `Element` are a `ModLattice`, and either a `Term` or the parameters to make a `Term`.

Class `EqRel`

To understand this class, it helps to know that Java does not have two-dimensional arrays like those in C or Pascal. Instead, it has arrays of arrays. The declaration

```
a = new int[5][10]
```

allocates space for 50 `ints` in 5 rows of 10, and also allocates a space for 5 references, which is filled with the references to these five rows. But then we can make an assignment like `a[1] = new int[20]`, and then the rows of `a` are no longer all the same length. We can also assign `a[2] = a[3]`, making two of the references point to the same space.

`EqRel` has the following fields:

`int size`: The `EqRel` represents an equivalence relation on the integers from 0 to `size - 1`.

`int [][] classes`: `classes[i]` holds the integers in the equivalence class of `i`. If `i` and `j` are in the same class, then `classes[i]` and `classes[j]` point to the same space, so the `EqRel` requires $O(\text{size})$ space.

The only parameter required to construct an `EqRel` is the `size`. Initially the `EqRel` holds the diagonal relation; afterwards the relation can be made coarser but not finer.

Note that the `EqRel` only manipulates numbers; it never sees the `Elements` that these numbers represent. The conversion between the numbers and the `Elements` is handled by the `ModLattice`.

User Interface

The interface is text-based and menu-driven: the program prints out a menu and asks the user to choose a command from the menu by typing in a number. Before each printing of the menu, the following information is displayed: the number of `Elements` in the active `ModLattice`, the row and column numbers of the next square to be filled, the size of the `ModLattice`, and the number of equivalence classes. (Only one `ModLattice` is active at a time.)

Menu options:

1. Generate new elements.

User will be prompted for a number, which should be greater than the current number of `Elements`, but not greater than the current size (it does not resize automatically, because the user may wish to repeatedly generate small numbers of new `Elements`, but it is inefficient to repeatedly resize by small increments). The program will fill squares in the `table` until either the `table` is full (which generates a special announcement) or the requested number of `Elements` has been reached.

2. Compute congruence.

If the `ModLattice`'s equivalence relation records that certain pairs of `Elements` are known to be equal, this command attempts to derive other equalities from these. It is not exhaustive, and using it twice in a row may produce equalities that were not produced the first time. The word "congruence" is used loosely here, as in general this calculation is not applied to the set of all elements of a lattice, but rather to a set of `Terms` representing (perhaps redundantly) a subset of its elements.

3. Impose a relation.

This command allows the user to force two `Elements` to be considered equal. The user will be prompted for the numbers of the two `Elements`.

4. Form quotient lattice.

This command causes the active `ModLattice` to be replaced by a new `ModLattice`, which is formed by discarding all but one `Element` from each equivalence class, and renumbering those that remain. "Quotient" is used loosely. If command 3 has not been used, then the new `ModLattice` represents the same lattice that

the old `ModLattice` represented, but if command 3 has been used, the new lattice is a homomorphic image of the old one.

The program keeps track of whether or not command 3 has been used. If it has, then in forming the new `ModLattice`, all information about conjugates and duality is discarded, because these features are not necessarily inherited by the quotient lattice.

5. Resize.

The user is prompted for a new size, and the `ModLattice`'s join/meet table and `Element` list are expanded to the new size. Repeated resizing by small increments is inefficient, so I recommend that you always resize by at least a factor of 1.5 unless this would exceed memory limitations.

6. Examine a square in the table.

The user is prompted for a row number and column number, and the program displays the number of the `Element` in this square of the join/meet table.

7. Examine a row in the join table or meet table.

The user is prompted to enter 1 for joins or 2 for meets, and to enter the number of an `Element` to form joins or meets with. The form of the output is the same as when the whole table is printed; see #9 below.

8. Examine an element.

The user is prompted for the number of the `Element`. The program displays the `Element`'s defining `Term` in both short and long form. For a generator, both forms are just the generator itself. For a join or meet, the short form is the numbers of the two arguments, with the appropriate symbol in between, while in the long form these numbers are replaced by the long form of the defining `Terms` of the arguments, yielding an expression involving only generators.

The program also displays the `Element`'s equivalence class, the short forms of its alternate `Terms`, the numbers of its conjugates, and, if it is in a `SelfDualModLattice`, the number of its dual. Any unknown information is given as "n/a".

9. Output information to a file.

This command is for printing information that is likely to be too big to fit on the screen. It prints a submenu, and prompts the user for a number and a filename. If the file already exists, the output will be appended to it. Option 1 in the submenu prints the `Element` list. For each `Element`, the program prints its number and both the short and long forms of its defining `Term`. Options 2, 3, and 4 print the join table, meet table, and join/meet table, respectively. (There is no join table or meet table stored in memory; this information is read from the join/meet table.) Each row of the table is a line of text, and for each square, the number of the `Element` in that square is printed, or "n/a" if the square is empty. At the beginning of each row the row number is given, separated from the row by a colon. Option 5 prints the equivalence relation. Each class is printed once, on one line of text. Option 6 allows the user to save to a file a complete description of the active `ModLattice`. The `ModLattice` can then be reconstructed from the file with command 13. The output of this option is not intended to be read by humans.

10. Change settings.

This command allows the user to change the symbolic constants that determine how much work is done in the computations. It displays the current value of the settings, displays a submenu, and prompts the user for a number. `DEPTH` is the depth of the recursion that the program uses when computing a square in the table. `DEPTH2` is used in place of `DEPTH` when the value of the square is already known. Both are initially set to 5. Increasing `DEPTH` or `DEPTH2` will exponentially increase the amount of time it takes to fill a square, and the base of the exponent depends on `ALTS`, which is the maximum number of alternative `Terms` an `Element` can store. `ALTS` is initially set to 6. `M` and `N` are used in computing congruences (see the description of the `computeCongruence()` method below.) `M` is initially set to 100, and `N` to 10.

11. Perform exhaustive check.

This command provides more tools for showing that `Elements` are equal. The submenu options are 1. Check join associativity, 2. Check meet associativity, 3. Check join/meet compatibility, and 4. Check modularity. (There is no need to check symmetry, because this is guaranteed by the way joins and meets are read from the table. So a `ModLattice` with a full table that passes all these tests is a bona fide modular lattice.) In options 1 and 2, for every triple of `Elements`, the program computes both sides of the identity, and if both sides are known and unequal, it records in the equivalence relation that they are equal. Option 4 similarly examines all triples that satisfy the appropriate inequality. (Note the program does not store the lattice's order relation; it knows $a \leq b$ only if either $a \vee b$ has been found to be b , or $a \wedge b$ has been found to be a .) Option 3 checks that $a \vee b = b$ if and only if $a \wedge b = a$; if one of these is found true but the other is not, the other is made true either by filling in an empty square or by recording the equality of two `Elements`.

Options 1 and 2 are very time-consuming, but 3 and 4 run quickly.

12. Use other method.

This command allows the use of code written by the user, without altering the program's code. (For example, the user might write a method to find and correct all failures of a certain identity.) The user is prompted for the name of a class and the name of a method in that class, and then the method is executed, with the active `ModLattice` as its argument. The method must be static, it must have exactly one parameter, and the type of this parameter must be `ModLattice` (not a subclass of `ModLattice`). The return type must be either `ModLattice` or void. If the method returns a `ModLattice`, the active `ModLattice` is replaced by the returned one.

13. Read lattice from file.

The user is prompted for the name of a file, which should contain the unmodified output of command 9 option 6 (see above). The active `ModLattice` is replaced with one constructed from the file.

0. Quit.

Quits.

Classes and methods

Class Driver

This class is a menu-driven user interface that allows the user to examine the active `ModLattice`, and call its essential methods. Class `Driver` is not meant to be instantiated; all of its members are static.

Fields:

`static ModLattice myLattice`: The active `ModLattice`.

`static boolean keepAuto`: Changes from `true` to `false` if the user uses command 3.

Methods:

`public static void start(ModLattice)`: A `ModLattice` must be passed to this method to get the program started. It is an infinite loop (it runs until the `System.exit()` command is reached when the user selects command 0). Each pass through the loop calls `printMenu()` to print the menu, `getLine()` to get the user's choice, and `actOn()`, explained below.

`private static String getLine(String)`: I got this method from one of my CS instructors. It displays its argument as a prompt, and returns the user's response.

`private static void actOn(String)`: This program converts its parameter into a number, and executes the command corresponding to the number, after prompting the user for any further input needed, and error-checking this input. Any exceptions resulting from bad input are caught, triggering an `InvalidCommandException` that transfers control back to `start()`.

`public static void printSubMenu1()`

`public static void actOnSub1(String)`

`public static void printSubMenu2()`

`public static void actOnSub2(String)`

`public static void printSubMenu3()`

`public static void actOnSub3(String)`: These methods handle the submenus generated in commands 9, 10, and 11.

`public static void outputAll(PrintStream)`: This is the code for command 9 option 6.

`public static void inputAll(BufferedReader)`: This is the code for command 13.

`public static String elementNumber(Element)`: Represents an `Element` in a form that can be displayed.

`public static String elementNumber2(Element)`:

A variant for making a machine-readable form, used in `outputAll()`.

`public static Element numberElement(String)`:

The inverse of `elementNumber2`, used in `inputAll()`.

Class Element

See Data Structures above for the description of the fields and constructors.

Methods:

`public String toString():` Produces the short form of the defining `Term`.

`public String fullName():` Produces the long form of the defining `Term`.

It is recursive; if the `Element` is a join or a meet, it puts the operation symbol between the `fullName()`s of the arguments, adding parentheses if necessary.

`public void addAlt(char, Element, Element)`

`public void addAlt(Term):` Stores a new alternate `Term`, if there is space for it and it is not equal to one already stored. The `Term` can be passed whole or in parts.

Class EqRel

See Data Structures above for a general description of the class and a description of the fields.

Methods:

`public boolean related(int, int):` Tells if the two arguments are in the same equivalence class.

`public void relate(int, int):` Merges the equivalence classes of the two arguments.

`public int [] classOf(int):` Returns the equivalence class of the argument.

`public int representative(int):` Returns the first `Element` of the equivalence class of the argument.

`public int [] repSet(IntCell count, int elementCount):` Returns a set of class representatives. The `ModLattice`'s `elementCount` is needed because the size of the `ModLattice` (which is also the size of the `EqRel`) may be greater than its `elementCount`, so some of the `EqRel`'s numbers do not represent `Elements`. This method ignores these extra numbers. The number of representatives is stored in the `IntCell` parameter (the number that was there before is ignored); this number may be less than the length of the array returned.

`public int countClasses(int elementCount):` Returns the number of equivalence classes. Again, numbers that do not represent `Elements` are ignored.

`void resize(int newSize):` Adds new numbers to the equivalence relation; all of the new numbers are initially in one-element classes.

Class IntCell

This class encloses a single `int` field called `myInt`. Unlike `java.lang.Integer`, the `IntCell`'s `int` is mutable. In this program, `IntCell` is used only to allow the method `EqRel.repSet()` to “return” two values: one as the return value, and one by modifying the `IntCell` parameter. (The other way to make a method “return” two values is to return an object that has both values as fields.)

Class `ModLattice`

See Data Structures above for description of the fields.

Methods:

`void fillCurrentSquare()`: This is the central method of the program. It computes the join or meet of the two `Elements` whose numbers are the current row and the current column. The operator is join if `row <= column`, and meet if `row >= column`.

Most of the work within `fillCurrentSquare()` is done by the 16 mutually recursive methods `handle1()` through `handle16()`. Each of these processes a different “term type”:

1. $a \vee b \vee c \vee d$
2. $a \vee b \vee (c \wedge d)$
3. $(a \wedge b) \vee (c \wedge d)$
4. $a \vee ((b \vee c) \wedge d)$
5. $a \vee (b \wedge c \wedge d)$
6. $a \vee b \vee c$
7. $a \vee (b \wedge c)$
8. $a \vee b$
9. $a \wedge b \wedge c \wedge d$
10. $a \wedge b \wedge (c \vee d)$
11. $(a \vee b) \wedge (c \vee d)$
12. $a \wedge ((b \wedge c) \vee d)$
13. $a \wedge (b \vee c \vee d)$
14. $a \wedge b \wedge c$
15. $a \wedge (b \vee c)$
16. $a \wedge b$

When a `handle()` method is called, this means that the join or meet being computed has been found to be equal to an expression of the corresponding type.

Note that 9 through 16 are the duals of 1 through 8 respectively. The code of `handle9()` through `handle16()` is dual to that of `handle1()` through `handle8()`.

All the `handle()` methods have void return type. They have the following parameters:

2, 3, or 4 **Elements** called **a**, **b**, **c**, and **d**, corresponding to the positions of **a**, **b**, **c**, and **d** in the above list of “term types”;

An **int** called **fresh**. At most one of **a**, **b**, **c**, and **d** can be “fresh” and subject to “recharacterization” (explained below). A value of 1 means **a** is fresh, 2 means **b** is fresh, 3 means **c** is fresh, 4 means **d** is fresh, and 0 means that none is fresh. Some of these methods never refer to this parameter, but it is in all of them for uniformity;

An **int** called **deep**. This is decreased by one each time another recursive call is made; when it is 0 the recursion stops; and

Two **Vectors** called **termResults** and **elementResults**. Each time the **handle9()** or **handle16()** is called, an appropriate **Term** is added to **termResults**, and each time the calculation reduces to a single **Element**, this **Element** is added to **elementResults**.

Each **handle()** method can perform various manipulations, depending on what relations **a**, **b**, **c**, and **d** satisfy, and whether certain joins and meets of them are known. There are 5 categories of manipulations: join, meet, recharacterize, absorb, and apply modular law.

Join: Replace two of **a**, **b**, **c**, and **d** with their join. For example, in **handle1()**, if the join of **b** and **c** is known to be **x**, **handle6()** is called with the **Elements** **a**, **d**, and **x**.

Meet: Replace two of **a**, **b**, **c**, and **d** with their meet; similar to join.

Recharacterize: One of the **Elements** **a**, **b**, **c**, and **d** is “fresh” if it was just created by a join or meet in the last **handle()** method called. Recharacterization means replacing the “fresh” element with one of its **Terms**. For example, in **handle6()**, if **c** is fresh and one of its **Terms** is $x \vee y$, then **handle1()** is called with **a**, **b**, **x**, and **y**; if another **Term** of **c** is $z \wedge w$, then **handle2()** is called with **a**, **b**, **z**, and **w**. In each **handle()**, only some of the parameters are allowed to be fresh, and the join and meet manipulations are designed to respect these restrictions.

Absorb: Simplify using one of the identities $(x \vee y) \wedge x = x$ and $(x \wedge y) \vee x = x$. This is the only manipulation that can yield a single **Element**. For example, in **handle5()**, if **a** is equal to **b**, **c**, or **d**, it is added to **elementResults**.

Apply modular law: for example, in **handle7()**, if **a** is known to be $\leq c$, then **handle15()** is called with **c**, **a**, and **b**.

When more than one of these manipulations is applicable, the method does all of them. This produces a tree of calls to **handle()** methods, and this tree can be very large. Sometimes the same **handle()** method will be called repeatedly with the same **Elements**; this yields redundant calculations. The program could have been written to eliminate this redundancy by using a **Hashtable** to keep track of the calls. I did not do this because I suspected that the time spent checking the **Hashtable** would be greater than the time saved by eliminating the repetition.

Now I explain how **fillCurrentSquare()** works. The two **Elements** that

are being joined or meteed are stored in the local variables `lhs` and `rhs`, and the operation is stored in the local variable `op`. First `handleTrivialCases()` is called. This method fills in the square in the cases when `lhs == rhs`, or when one of them is 0 or 1. If one of these cases applies, `handleTrivialCases()` returns `true`, and the rest of `fillCurrentSquare()` is skipped.

Otherwise it next calls `preparation()`. This method first checks to see if the square is already filled; if so, the `Element` there is added to `elementResults`. Then it tries to compute the join or meet via the automorphisms. If any of these computations succeed, the results are added to `elementResults`.

Next `handleEasyCases()` is called. This method detects certain cases in which the answer is immediately given by the absorption identities; any answer found here is also added to `elementResults`.

Next `dispatchOnOps()` is called, which calls the appropriate `handle()` method depending on whether the defining `Terms` of `lhs` and `rhs` are joins, meets, or generators. The `deep` parameter of the `handle()` method is initially assigned `DEPTH` if `elementResults` is empty, and `DEPTH2` if it is not. The `handle()` methods can add `Terms` to `termResults` and `Elements` to `elementResults`.

Next comes `turnTermsToElements()`. For each `Term` in `termResults`, this method checks to see if its value is given in the table. If so, the `Element` found is added to `elementResults`; otherwise, the `Term` is added to a `Vector` called `unknownTerms`.

If after all this work `elementResults` is still empty, `makeNewElement()` (not to be confused with `makeNewElements()`) is called to create the new `Element`. If `elementResults` contains more than one `Element`, `identify()` is called to record that they are equal.

Finally the square is filled, and `fillOtherSquares()` is called. Each `Term` in `unknownTerms` gives a join or meet that has been proved equal to the `Element` in the current square, so this `Element` is also placed in the squares for these joins and meets. Also, these `Terms` are installed as the `Element`'s alternate `Terms`, but only if both of their arguments are `Elements` with lower numbers. We only want to describe `Elements` in terms of simpler `Elements`, and having a lower number corresponds approximately to having a shorter `Term`.

Other methods in class `ModLattice`:

`public void makeNewElements(int total)`: This is command 1 in the menu. Note that the parameter is the total to be reached, not the number of new `Elements` to be created. This method repeatedly calls `fillCurrentSquare()` to fill the current square of the table, and `updatePointer()` to find the next square, until either the required number of `Elements` have been made, or the table is filled (in which case it announces that the lattice is complete.) Then it calls `computeAllConjugates()`.

`protected Element join(Element, Element)`

`protected Element join(int, int)`

`protected Element meet(Element, Element)`

`protected Element meet(int, int)`: These methods are for looking up information in the table, not for computing new joins and meets. Either the

Elements or their numbers may be used as arguments. Perhaps there should also be methods that accept one argument of each type. These would be useful, but they seem like bad style to me.

`protected void identify(Element, Element)`: Records the equality of these two Elements.

`public boolean computeConjugate(Element e, int i)`:

This method attempts to compute the effect of automorphism `i` on Element `e`, returning true if successful. It uses the defining Term of `e` first, and then any alternate Terms. The answer is recorded in `e.conjugates[i]`. If a different answer is already recorded there, `identify()` is called. This method uses the helper `computeConjugate(Term, int)`.

`public void computeAllConjugates()`: Calls `computeConjugate()` with every Element and every automorphism.

`void computeCongruence()`: This method is command 2 in the menu. It has two parts. In the first part, for every two Elements that are related, their join and meet are looked up; if either of these is defined, it is made related to the two Elements. The second part depends of the constants M and N. The first M nontrivial equivalence classes are examined, and for each of these classes the first Element is compared to each of the next N. For the two Elements being compared, the corresponding rows of the join table are examined square by square. Any discrepancies are fixed by either filling in empty squares or calling `identify()`. The same operation is done with the meet table.

`public ModLattice collapse(boolean)`: This is command 4 in the menu. If the argument is false, all information about automorphisms is left out of the new ModLattice. This method creates a new ModLattice, with newly created Elements corresponding to a set of class representatives of the old ModLattice. Note that every Element contains references to other Elements; if a class representative contains a reference to Element `x`, then the corresponding Element in the new ModLattice gets a corresponding reference to the new Element that corresponds to the class representative of `x`.

`void resize(int newSize)`: This is command 5 in the menu. The information in the join/meet table and the Element list is copied into larger newly allocated arrays, and the EqRel is resized.

`void forceJoin(int, int)`

`void forceMeet(int, int)`: These methods are not available from the menu; they can only be invoked by user-written code. They force the join or meet of two Elements (whose numbers are the arguments) to be computed, deviating from the order in which the table is normally filled. In most situations these methods should not be used, because they cause Elements to be numbered in unusual ways. They are necessary for some applications, because one must create Elements before one can identify them, but sometimes creating them in the usual sequence would take too long or use too much space.

If `elementCount == currentSize` when the user attempts to use one of these two methods to fill an empty square in the table, the program will crash. Any code that invokes `forceJoin()` or `forceMeet()` should have a safeguard to prevent this.

Class `SelfDualModLattice`

This class is a subclass of `ModLattice`. It has methods `computeDual()` and `computeAllDuals()` that are analogous to `ModLattice`'s `computeConjugate()` and `computeAllConjugates()`. It overrides the `ModLattice` methods `makeNewElements()`, `preparation()`, `makeNewElement()`, and `collapse()`, in each case adding instructions to deal with duals, which are analogues of the instructions that deal with conjugates. In `collapse()`, if the argument is `true`, a new `SelfDualModLattice` is constructed, retaining the information about duals, but if the argument is `false`, a `ModLattice` that is not a `SelfDualModLattice` is returned, without this information.

Class `Term`

Methods:

`public boolean equals(Term)`: Two `Terms` are considered equal if they have the same operation and the same two arguments, even if the order of the arguments is reversed.

`public String toString()`: If this `Term` is a join or a meet, this method returns the operation symbol in between the numbers of the two arguments; otherwise it returns a `String` consisting of the generator.

Performance issues

Because this program does not stop computing when it has found an answer, increasing its computational capability (by using automorphisms and duality, by increasing `DEPTH`, `DEPTH2`, or `ALTS`, or by modifying the code) makes it run slower, though it should yield better results. However, better results can save time. For example, one might use command 1 to find the next 100 elements, and then find after further computation that those 100 reduce to only 6 distinct new elements. One might have to repeat this procedure 15 times to find the desired 100 new elements. If one could compute more slowly and find the 100 distinct elements on the first try, that might be faster. Also, using more computational power may increase the number of squares that are filled by `fillOtherSquares()`; this will save time later if `DEPTH2` is set lower than `DEPTH`.

Possible modifications and extensions

The manipulations used in the `handle()` methods are not the only possible ones. There were some that I deliberately left out because I thought they would primarily lead to redundant calculations. There may be other useful ones that did not occur to me. Feel free to add and remove manipulations as you see fit. Other `handle()` methods could also be added to deal with longer expressions. `dispatchOnOps()` could be modified to dig deeper into `lhs` and `rhs`, and start the computation with a longer expression.

One could try using a `Hashtable` to eliminate redundant calls to the `handle()` methods. This might speed up execution, especially for higher values of `DEPTH`.

I have said that this program is for studying modular lattices, but it could easily be modified to handle nonmodular lattices by removing all use of modularity in the code. Modularity is used in only two places: the “apply modular law” manipulations in the `handle()` methods, and the “Check modular law” command in the `Driver`.

Other properties of lattices, such as the semidistributive laws, could be incorporated as new manipulations in the `handle()` methods. (If you want to compute in distributive lattices, I think a much simpler program than this one would suffice.) Properties involving longer expressions, such as the Arguesian identity, could be incorporated in the same way if the appropriate `handle()` methods were included. However, I think it would be more efficient to deal with such properties by writing separate methods (invoked through command 12) to search for and correct failures.

This program has no data structure that directly represents the order relation of a lattice. There are many places in the code where the program needs to know that one `Element` is less than another, and currently it can only determine this if the join or meet of the two `Elements` has been computed. If this data structure were added, there would be ways that comparabilities could be determined, stored, and used before the join and meet were computed.

It should be easy to modify `Driver` to have multiple active `ModLattices`. Perhaps a class called `Homomorphism` could be designed to represent homomorphisms between the active `ModLattices`.

Some changes could be made in the way alternate `Terms` are stored in `Elements`. Currently, an `Element` only stores `Terms` that involve two lower-numbered `Elements`, because `Elements` with lower numbers are presumed to be simpler (this criteria is encoded in `ModLattice`’s `fillOtherSquares()` method, which calls `Element`’s `addAlt()` method). Instead, every `Term` could be made to compute its own length, and length could be used as the criterion for simplicity. Also, currently the `addAlt()` method accepts the first `ALTS Terms` that meet the criterion. Instead, it could be willing to discard a `Term` in order to accept one that is better in some sense. “Better” might mean that it is shorter, or that it has lower joinands or higher meetands. Also, if an `Element`’s `Terms` are all joins, it would be useful to replace one of them with a meet.